

Constraint-Based Locality Analysis for X10 Programs

Qiang Sun^{1,3,4} Yuting Chen^{2,4} Jianjun Zhao^{2,3}

¹Department of Computer Science & Engineering and ²School of Software, Shanghai Jiao Tong University, Shanghai, China

³State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

⁴Shanghai Key Laboratory of Computer Software Testing & Evaluating, Shanghai, China

sun-qiang@sjtu.edu.cn, chenyt@cs.sjtu.edu.cn, zhao-jj@cs.sjtu.edu.cn

Abstract

X10 is a HPC (High Performance Computing) programming language proposed by IBM for supporting a PGAS (Partitioned Global Address Space) programming model offering a shared address space. The address space can be further partitioned into several logical locations where objects and activities (or threads) will be dynamically created. An analysis of locations can help to check the safety of object accesses through exploring which objects and activities may reside in which locations, while in practice the objects and activities are usually designated at runtime and their locations may also vary under different environments. In this paper, we propose a constraint-based locality analysis method called *Leopard* for X10. *Leopard* calculates the points-to relations for analyzing the objects and activities in a program and uses a place constraint graph to analyze their locations. We have developed a tool to support *Leopard*, and conducted an experiment to evaluate its effectiveness and efficiency. The experimental results show that *Leopard* can calculate the locations of objects and activities precisely.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Concurrent, distributed, and parallel languages; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

General Terms Languages, Analysis

Keywords Locality Analysis, Points-to Analysis, Concurrency

1. Introduction

Partitioned Global Address Space (PGAS) is a parallel programming model offering a shared address space [23]. The shared address space can be further partitioned into several logical places, where activities (or threads) can be dynamically created or enabled, and objects can be dynamically allocated or copied. Some PGAS programming languages such as X10 proposed by IBM also require the activities not directly access the remote objects, thus their compilers usually make some runtime safety checks for object access operations [4].

Locality analysis is an analysis technique that can help check the safety of object accesses in PGAS programs. It explores which objects and activities may reside in which locations and identifies

whether an activity accesses the field of a remote object. However, locality analysis usually faces some challenges in practice. One main challenge is that objects or activities usually need to be dynamically created, and their places may be designated at runtime (e.g., a place expression is used to assign a place an activity resides in), while the space partitions can vary under different environments. Thus it is not easy, if not impossible, to statically infer the place in which objects and activities reside. In addition, some PGAS programming languages, such as X10, support cyclic places, which denote that each place has a predecessor and a successor and all places are logically organized in a circle. It also sets barriers to locality analysis in that places may be designated and evaluated through using expressions.

In this paper we propose a constraint-based locality analysis called *Leopard* (Locality Analyses of Partitioned Global Address Space Programs) for X10 programs. We first define a subset-based constraint system to describe the constraints on the objects, activities, and their places, and then calculate the possible values of each place expression and model the queries on all places. *Leopard* also analyzes the points-to relations in the program and uses a place constraint graph (PCG) to achieve the places of the objects and activities. We also take an activity- and heap-sensitive analysis of locality information in order to improve the precision of analysis.

The paper makes the following contributions:

- **Abstraction.** We define the places taking into account different topology structures of places, query operations (i.e., place expressions), and a locality model which can be specialized for a specific topology structure. We also define the satisfiability relation between the locality model and a set of constraints on the program.
- **Algorithm.** We present a context-insensitive algorithm and a context-sensitive one to achieve the constraints on objects and activities. Especially, our context-sensitive locality analysis trades off between the cost and precision through choosing different context lengths.
- **Tool support and experiments.** We have developed a tool to support *Leopard*, and conducted two experiments to measure its performance and precision. The experimental results show that *Leopard* can be used to analyze the locality information precisely.
- **Application.** The locality information computed by *Leopard* can be used to infer place equivalences with respect to different topology structures and check safe accesses to objects in X10 programs. We have implemented an application to utilize the locality information to verify safe accesses to instance fields, and also conduct an experiment to compare it with a place type system [4]. The result shows that *Leopard* provides with more support in checking safe accesses.

```

1 public class T {
2     var x:int;
3     def this(n:int){x=n;}
4 }
5 public class BoxT {
6     var data:T;
7     public static def main(args:Rail[String]): Void {
8         val b = new BoxT(); // creating ol8
9         val bc = b;
10        val t = new T(0); // creating ol10
11        val p:Place = b.home();
12        val q = p.next();
13        async(q.next()){
14            val r = new T(1); // creating ol14
15            b.data = r;
16            Console.OUT.println("data@[Child]: "+b.data.x);
17        }
18        bc.data = t;
19        Console.OUT.println("data@[Root]: "+b.data.x);
20    }
21 }

```

Figure 1. A Sample X10 Program

The remainder of the paper is organized as follows. Section 2 presents an example to illustrate how the analysis algorithm works. Section 3 presents the foundation of context-insensitive locality analysis. Section 4 presents the details of the algorithm for the context-insensitive locality analysis of X10 programs. Section 5 presents the activity-sensitive and heap-sensitive locality analysis. Section 6 describes an experiment to evaluate the performance, precision and usability of Leopard. Section 7 discusses the related work. Section 8 concludes this paper.

2. Background and Example

2.1 Place and Activity

An X10 program consists of activities executing in places plus the ability to access the remote data by creating and accessing references to the data. We next describe two language constructs satisfying the X10 specification (v2.0) [22] which needs to be focused on during our analysis.

Places. In X10, a place is a collection of resident activities and objects. Programmers can use the configuration options to set the number of places. Thus all places can be initialized before the program is executed. X10 has a type *Place* which denotes a certain place of execution. Values of a place-typed variable can be obtained in the following ways:

- *here* denotes a place where an activity is executing.
- Given a reference variable *v*, *v.home()* is a method invocation that returns the value of the place where the object that *v* points to resides.
- X10 organizes the places of a program through using a cycle. Given a place *p*, *p.next()* returns its successor, and *p.prev()* returns its predecessor.

Asynchronous Activities. X10 supports asynchronism through spawning and executing activities: an activity can dynamically spawn other activities in local or remote places, e.g., *async(p) S* creates an activity to execute the statement *S* in place *p*; once an activity (resp. object) is created in a place, it cannot migrate to another place; an activity cannot access a remote object directly, but needs to spawn remote activities to access the remote objects; a root activity is spawned in *Place(0)* by default and it corresponds to invoke the main method.

Figure 1 shows an example of X10 program containing two classes, *T* and *BoxT*, where class *T* has a field *x* of the type *int*, and class *BoxT* has a field *data* of the type *T*. An execution of the program creates a root activity which invokes the main method in

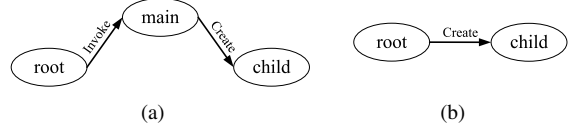
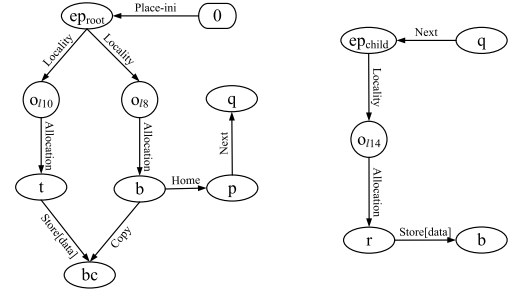


Figure 2. The ANG of the Program in Figure 1



(a) PCG of the Root Activity. (b) PCG of the Child Activity.

(c) PCG of the Sample Program.

Figure 3. PCGs of the Program in Figure 1. (a) and (b) are intraprocedural PCGs. (c) is an interprocedural PCG.

class *BoxT*, and spawns a child activity (see lines 13-17) to update *b.data*. Note that the child activity is created and enabled in the place assigned by a place expression *q.next()*. In this example, *b.data = r*; (line 15) may cause an access violation if *b* points to a remote object.

2.2 An Illustrative Example

We use the program in Figure 1 to illustrate how Leopard works. Especially, we achieve different locality graphs for different topology structures of places in the analysis.

The first step of our locality analysis is to build an activity nesting graph (ANG) for the program, where the nodes of the ANG represent the methods and activities in the program and the edges represent the method invocations and activity creations. As Figure 2(b) shows, the ANG contains two nodes representing the root and the child activities, and one edge from the root node to the child node representing the creation of the child activity. Since the root activity invokes only one method *main()*, we omit the node for the *main()* method of Figure 2(a).

The second step is to build a place constraint graph (PCG) for each activity or method in the ANG. A PCG describes the creations of objects, points-to assignments, and place expressions. Figures 3(a) and 3(b) represent the PCGs for the root and the child, respectively. The node *ep_root* (resp. *ep_child*) represents the place in which the root activity (resp. the child one) resides. In addition, the root activity is spawned in *Place(0)* by default.

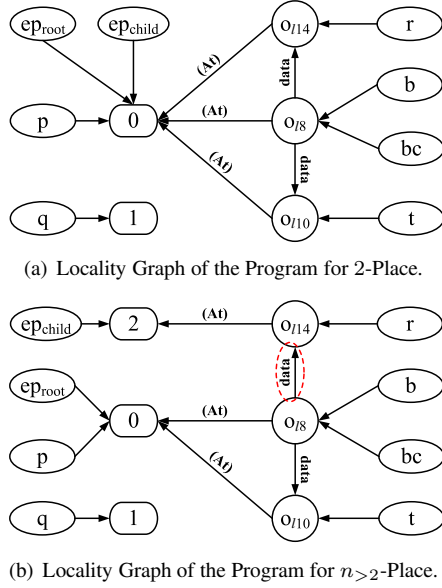


Figure 4. Locality Graphs.

The third step is to combine the intraprocedural PCGs into an interprocedural PCG to provide a comprehensive graph about the locality information and then facilitate the calculation of the places of objects and activities. During the combination, any variable accessed by different activities will be merged. For example, the variable q can be accessed by both the root and the child activities, as Figures 3(a) and 3(b) show. A PCG after combination is shown in Figure 3(c).

The fourth step is to solve the interprocedural PCG and get a locality graph, which explicitly provides all locality information in the program. A locality graph is initialized as a graph containing all nodes in the PCG. Solving the interprocedural PCG involves generating a locality graph, each edge of which is an arrow from an activity or object to a place and represents that the activity or object is created in the place. In the example we achieve different locality graphs for different cyclic topologies of the places:

- If the number of places is one, all the activities and objects are created in the same place. Therefore, all the accesses to the objects are safe.
- If the number of places is two, the locality graph for the program is given in Figure 4(a). In this figure, both ep_{root} and ep_{child} point to $Place(0)$, which indicates that the root and the child activities are all spawned in $Place(0)$. The objects O_{18} , O_{10} and O_{14} are all created in $Place(0)$, and the variable bc is an alias of the variable b because they both point to O_{18} ;
- If the number of the places is greater than two, the locality graph is shown in Figure 4(b), where the root and the child activities are executed in the first and the third places (according to the edges $ep_{root} \rightarrow 0$ and $ep_{child} \rightarrow 2$), respectively. The child activity may either write the copy of O_{18} which resides in the third place (see line 15), or throw an exception named **BadPlaceException**.

3. Foundation of Locality Analysis

3.1 X10 Program Representation

We next define some program notations following by three abstract domains representing three important program properties of X10

language (v2.0) [22]. These properties are used to capture the queries of places and object-oriented features.

A program in X10 can be represented as a collection of class types $Class$, which refer to the classes in the program.

A set F is defined to include all non-static fields, each of which can be accessed using the form $v.f$, where v is a variable with a class type. F can be divided into a set F_o containing the fields of class type and a set F_p containing the fields of place type.

A set M is defined to include all class methods.

A set R is defined to include all local variables, formal parameters of methods in M and static fields in the classes. R can be divided into a set P containing the variables of place type (and the auxiliary variables introduced by the analysis) and a set V containing the variables of class type.

A set A is defined to include all activity names. Since an activity is created and its place may be evaluated at runtime, we also define a mapping function to trace all potential places with respect to an activity or method

$$\mu : A \cup M \rightarrow P$$

For example, given the program in Figure 1, we have $\mu(a_{root}) = ep_{root}$ and $\mu(a_{child}) = ep_{child}$.

A set O is defined to include the names of all objects created by the object allocation statements (e.g., $v = new C()$). In a context-insensitive analysis, each object name corresponds to one allocation statement.

Let MP be the maximum number of places designated at configuration time. A set PC is defined to model the place constants

$$PC = \{0, 1, \dots, MP - 1\}.$$

In order to take an MP -independent analysis, we also define a set PN to include all *place names*. Each place name is specified by using an integer, and instantiated to a place constant at runtime by

$$Eval_{MP} : PN \rightarrow PC \triangleq \lambda z.z \bmod MP,$$

where \bmod denotes the modulo operation.

For a given name z , the query of its next (resp. previous) n^{th} place can be achieved by $z + n$ (resp. $z - n$). For a name set $S_{pn} \subseteq PN$, we define the right and the left shift queries by

$$S_{pn} \gg n = \{z + n | z \in S_{pn}\}, \quad S_{pn} \ll n = \{z - n | z \in S_{pn}\}.$$

Especially, we use \top to denote the upper bound of the place name sets. We have

$$\top \gg n = \top, \quad \top \ll n = \top.$$

We use $\wp(S)$ to denote the powerset of a set S . Three abstract domains are defined for modeling the program properties (e.g., locality information and points-to relations):

- The points-to analysis computes points-to relations between variables of pointer types and their allocation sites. There are two kinds of pointer variables:

- variable $v \in V$, and
- object field $o.f$, where $o \in O$ and $f \in F_o$.

Then a points-to relation function is

$$\rho \in AbstractReference = (V \cup (O \times F_o) \rightarrow \wp(O)).$$

- One object allocation statement may create an object in different places at runtime. In order to track the objects created by one statement in different places, we define

$$\sigma \in AbstractObject = (O \rightarrow \wp(PN)).$$

- In order to model the relationships between the place-typed variables and their values, we define

$$\theta \in AbstractPlace = (P \cup (O \times F_p) \rightarrow \wp(PN)).$$

[seq]	(ρ, σ, θ)	$\models_I s_1 s_2$ iff $(\rho, \sigma, \theta) \models_I s_1 \wedge (\rho, \sigma, \theta) \models_I s_2$
[if]	(ρ, σ, θ)	$\models_I \text{if}(\text{cond}) s_1 \text{ else } s_2$ iff $(\rho, \sigma, \theta) \models_I s_1 \wedge (\rho, \sigma, \theta) \models_I s_2$
[async]	(ρ, σ, θ)	$\models_I \text{async}(p) s$ iff $(\rho, \sigma, \theta) \models_I s \wedge \theta(p) \subseteq \theta(\mu(a))$
[copy]	(ρ, σ, θ)	$\models_I v_1 = v_2$; iff $\rho(v_2) \subseteq \rho(v_1)$
[obj]	(ρ, σ, θ)	$\models_I v = \text{new } C();$ iff $\{o\} \subseteq \rho(v) \wedge \theta(\mu(\text{mdl})) \subseteq \sigma(o)$
[load]	(ρ, σ, θ)	$\models_I v_1 = v_2.f$; iff $\forall o \in \rho(v_2), \rho((o, f)) \subseteq \rho(v_1)$
[store]	(ρ, σ, θ)	$\models_I v_1.f = v_2$; iff $\forall o \in \rho(v_1), \rho(v_2) \subseteq \rho((o, f))$
[ini]	(ρ, σ, θ)	$\models_I p = \text{Place}(i)$; iff $\{i\} \subseteq \theta(p)$
[home]	(ρ, σ, θ)	$\models_I p = v.\text{home}()$; iff $\forall o \in \rho(v), \sigma(o) \subseteq \theta(p)$
[here]	(ρ, σ, θ)	$\models_I p = v.\text{here}$; iff $\theta(\mu(\text{mdl})) \subseteq \theta(p)$
[ass]	(ρ, σ, θ)	$\models_I p_1 = p_2$; iff $\theta(p_2) \subseteq \theta(p_1)$
[np]	(ρ, σ, θ)	$\models_I p_1 = p_2.\text{next}()$; iff $\theta(p_2) \gg 1 \subseteq \theta(p_1)$
[pp]	(ρ, σ, θ)	$\models_I p_1 = p_2.\text{prev}()$; iff $\theta(p_2) \ll 1 \subseteq \theta(p_1)$
[pl]	(ρ, σ, θ)	$\models_I p = v.f$; iff $\forall o \in \rho(v), \theta((o, f)) \subseteq \theta(p)$
[ps]	(ρ, σ, θ)	$\models_I v.f = p$; iff $\forall o \in \rho(v), \theta(p) \subseteq \theta((o, f))$
[inv]	(ρ, σ, θ)	$\models_I r = r_0.m(r_1, \dots, r_k);$ iff $\varphi(\text{ret}_m, r) \wedge \varphi(r_0, \text{this}_m) \wedge \varphi(r_1, fr_1) \wedge \dots$ $\wedge \varphi(r_k, fr_k) \wedge \theta(\mu(\text{mdl})) \subseteq \theta(\mu(m))$

Figure 5. Locality Analysis Specification

3.2 Locality Analysis Specification

The principle of context-insensitive locality analysis is to define an abstract specification describing a system of constraints and a domain

$$\text{AbstractReference} \times \text{AbstractObject} \times \text{AbstractPlace}$$

and to calculate the least element of the domain that satisfies the constraints. The result of the locality analysis can be defined by a tuple (ρ, σ, θ) . The formulation of the abstract specification is in the form of

$$(\rho, \sigma, \theta) \models_I s$$

where s is a statement in an X10 program and \models_I represents the largest acceptable relation that satisfies the abstract specification. \models_I is defined in the clauses shown in Figure 5.

[seq], **[if]** and **[async]** are clauses related to control flows. Each statement within a control structure must be analyzed in a consistent way using (ρ, σ, θ) . **[seq]** and **[if]** define the acceptable analysis result for the structures of sequence and branch, respectively. These two structures have the same semantics as those in Java programs. **[async]** defines the acceptable analysis result for activity creation. In **[async]**, a is the activity created by $\text{async}(p) s$, and $\mu(a)$ is the associated place-typed variable.

[copy], **[obj]**, **[load]** and **[store]** are clauses for points-to analysis. These four clauses are inspired by the algorithm of Andersen [3]. Compared with the equality-based analysis [17], Andersen's analysis approach is more precise due to its one-way propagation of points-to sets [7]. **[copy]** says that a copy statement $v_1 = v_2$ makes the points-to set of v_2 be a subset of that of v_1 . **[obj]** says that an object creation statement $v = \text{new } C()$ allocates a new object o of type C , makes o be an element of the points-to set of v , and records the place names of the object o . Here mdl represents the current activity or method. **[load]** says that a load statement $v_1 = v_2.f$ makes the points-to set of $o.f$ be a subset of that of v_1 , where o is the object in the points-to set of v_2 . Similarly, **[store]** says that a store statement $v_1.f = v_2$ makes the points-to set of v_2 be a subset of that of $o.f$, where o is the object in the points-to set of v_1 .

[ini], **[home]**, **[here]**, **[ass]**, **[np]**, and **[pp]** are clauses for place expressions. **[ini]** says that a statement $p = \text{Place}(i)$ adds the i^{th} place into the place set of the place-typed variable p . **[ass]** says that a statement $p_1 = p_2$ makes the place set of p_2 be a subset of that of p_1 . **[home]** says that a statement $p = v.\text{home}()$ makes the place set of p contain the place in which the object o pointed to by v is

created. **[here]** says that a statement $p = \text{here}$ makes the place set of p contain the place in which the current activity (resp. method) is running. **[np]** says that the statement $p_1 = p_2.\text{next}()$ makes the place set of p_1 contain the next places in that of p_2 . **[pp]** says that a statement $p_1 = p_2.\text{prev}()$ makes the place set of p_1 contain the previous places in that of p_2 .

Any place-typed variable p can access a place-typed field f . **[pl]** says that a place-load statement $p = v.f$ makes the place set of $o.f$ be a subset of that of p , where o is the object in the points-to set of v . Similarly, **[ps]** says that a place-store statement $v.f = p$ makes the place set of p be a subset of that of $o.f$.

[inv] formulates the constraints generated by analyzing a method invocation statement. These constraints characterize the parameter passing from the actual parameters of the call site to formal parameters of the target method. In the clause, ret_m is the return value of the method m , $fr_i (1 \leq i \leq k)$ is the i^{th} formal parameter of the method m , and φ is

$$\varphi(a, b) = \begin{cases} \rho(r_a) \subseteq \rho(r_b) & r_a, r_b \in V, \\ \theta(r_a) \subseteq \theta(r_b) & r_a, r_b \in P, \\ \text{false} & \text{otherwise.} \end{cases}$$

This clause is necessary in the interprocedural analysis.

4. Context-Insensitive Locality Analysis Algorithm

Our context-insensitive locality analysis includes constructing an activity nesting graph for the objective program as well as its place constraint graph, and then solving the place constraint graph.

4.1 Building Activity Nesting Graph

An activity nesting graph (ANG) describes the spawns of activities and invocations of methods in an X10 program. An ANG can be formalized as $(M \cup A, E_{ang})$, where E_{ang} contains all the edges in the form $n_1 \rightarrow n_2$, where $n_1, n_2 \in M \cup A$. Thus an edge $e \in E_{ang}$ can be either an activity creation edge whose target is an activity node or a method invocation edge whose target is a method node.

An ANG is constructed by traversing the statements of the form $\text{async}(p) s$ or $r.m()$. When an activity a' (a statement $\text{async}(p) s$) is defined in a method m (resp. in an activity a), an activity creation edge from m (resp. a) to a' is added into the ANG. When a method call $r.m'()$ is included in a method m (resp. in an activity a), a method invocation edge from m (resp. a) to m' is added into the ANG. If a method invocation is polymorphic, the edges from the caller to all potential target methods are added into the ANG.

4.2 Building Place Constraint Graph

A place constraint graph (PCG) captures the constraints on the statements in a method or activity. A node in a PCG can be

- a variable node representing a variable in V ,
- an object node representing an object in O ,
- a place-typed variable node representing a place-typed variable in P , or
- a place node representing a place name in PN .

An edge represents a constraint on nodes and can be

- an allocation edge $o \in O \rightarrow v \in V$ in the set E_A ,
- a copy edge $v_1 \in V \rightarrow v_2 \in V$ in E_C ,
- a load edge $v_1 \in V \xrightarrow{f \in F_o} v_2 \in V$ in E_L ,

Statement	Edge Generation	Description
$v_1 = v_2$	$(v_2 \rightarrow v_1) : E_C$	Add a copy edge to E_C .
$v = \text{new } C()$	$(o \rightarrow v) : E_A,$ $(ep, o) : E_{loc}$	Add an allocation edge to E_A and add a locality edge to E_{loc} .
$v_1 = v_2.f$	$(v_2 \xrightarrow{f} v_1) : E_L$	Add a load edge to E_L .
$v_1.f = v_2$	$(v_2 \xrightarrow{f} v_1) : E_S$	Add a store edge to E_S .
$p = \text{Place}(i)$	$(i \rightarrow p) : E_{PI}$	Add an edge $(i \rightarrow p)$ to E_{PI} .
$p = v.\text{home}()$	$(v \rightarrow p) : E_H$	Add a home edge to E_H .
$p = \text{here}$	$(ep \rightarrow p) : E_{PC}$	Add a place-copy edge to E_{PC} .
$p_1 = p_2.\text{next}()$	$(p_2 \rightarrow p_1) : E_N$	Add a next-place edge to E_N .
$p_1 = p_2.\text{prev}()$	$(p_2 \rightarrow p_1) : E_P$	Add a prev-place edge to E_P .
$p = v.f$	$(v \xrightarrow{f} p) : E_{PL}$	Add a place-load edge to E_{PL} .
$v.f = p$	$(p \xrightarrow{f} v) : E_{PS}$	Add a place-store edge to E_{PS} .

Table 1. Generating Place Constraint Edges

- a store edge $v_1 \in V \xrightarrow{f \in F_o} v_2 \in V$ in E_S ,
- a place-initialization edge $pn \in PN \rightarrow p \in P$ in E_{PI} ,
- a place-load edge $v \in V \xrightarrow{f \in F_p} p \in P$ in E_{PL} ,
- a place-store edge $p \in P \xrightarrow{f \in F_p} v \in V$ in E_{PS} ,
- a place-copy edge $p_1 \in P \rightarrow p_2 \in P$ in E_{PC} ,
- a next-place edge $p_1 \in P \rightarrow p_2 \in P$ in E_N ,
- a previous-place edge $p_1 \in P \rightarrow p_2 \in P$ in E_P ,
- a home edge $v \in V \rightarrow p \in P$ in E_H , or
- a locality edge $p \in P \rightarrow o \in O$ in E_{loc} .

For example, Figures 3(c) shows the PCG of the program in Figure 1.

Intraprocedural PCG Construction. Since interleavings exist in the execution of parallel activities, the number of traces may explode even in a small X10 program. In order to reduce the analysis cost, Leopard adopts a flow-insensitive algorithm for building the intraprocedural PCG. One major difference between the flow-sensitive and flow-insensitive algorithms is that the former computes the solutions at program points through flows, and the latter does not propagate the computation through control flows, but computes a single solution for a program or a method. An intraprocedural PCG is constructed for an activity (resp. a method) to contain the nodes in V , O , P , and PN . A place-typed variable ep is generated for each activity (resp. method) indicating the place in which the activity (resp. method) is executed. The place constraint edges are generated by adopting the rules in Table 1. For example, when the statement `var p:Place=b.home();` is analyzed, a home edge $b \rightarrow p$ is added into the set E_H .

Interprocedural PCG Construction. An interprocedural analysis is performed through using an ANG to combine the PCGs of all activities and methods into one graph (i.e., interprocedural PCG). During the analysis, all edges in the ANG are traversed, and the interprocedural PCG is constructed.

- When an activity creation edge $n \rightarrow a$ is visited, since the activity a can access some variables in n and some nodes of two PCGs can represent the same variable, two PCGs can be ‘connected’ by merging these nodes. In addition, if a has the form `async(p) s`, the algorithm generates a place-copy edge $p \rightarrow ep_a$. For example, Figures 3(a) and 3(b) are the PCGs of two activities *root* and *child*, respectively. Since an activity creation edge *root* \rightarrow *child* exists in the ANG, the PCGs can be combined by merging the nodes q and b .

- When a method invocation edge is visited, the PCGs of the caller and callee can be ‘connected’ by adding copy edges from the actual parameters to the formal ones as well as from the return value to the corresponding variable in the caller. This procedure is formulated by the clause [inv] in Figure 5. A place-copy edge from ep_{caller} to ep_{callee} also needs to be added in order to transfer the place information between the activities holding the caller and callee.

4.3 Solving a Place Constraint Graph

Once the PCG is constructed, we take two steps to compute a solution (i.e., the locality information of a program): points-to analysis and place-value analysis. The specific topology structures of the program can help to achieve the detail locality information, but not be necessary in solving the locality.

Firstly, the points-to relations can be computed by the worklist algorithm of SPARK [13]. The algorithm builds up allocation relationships and initializes *worklist* maintained by the solver, which contains variable nodes whose points-to objects need to be propagated. Next, the algorithm deals with the worklist and all the field store/load edges iteratively to obtain the final solution. When a variable node v is included in the worklist, its points-to set needs to be propagated along the edges of the PCG. If the points-to set of v is modified during the propagations, v is added to the worklist.

Algorithm 1: Place Sets Propagation

```

1 begin
2   foreach  $(i \rightarrow p) \in E_{PI}$  do
3     propagate sets  $\{i\} \rightarrow \theta(p)$ 
4   repeat
5     foreach  $(p_1 \rightarrow p_2) \in E_{PC}$  do
6       propagate sets  $\theta(p_1) \rightarrow \theta(p_2)$ 
7     foreach  $(p \rightarrow o) \in E_{loc}$  do
8       propagate sets  $\theta(p) \rightarrow \sigma(o)$ 
9     foreach  $(v \rightarrow p) \in E_H$  do
10      foreach  $o \in \rho(v)$  do
11        propagate sets  $\sigma(o) \rightarrow \theta(p)$ 
12     foreach  $(p_1 \rightarrow p_2) \in E_N$  do
13       if  $\theta(p_1) == \top$  then
14          $\theta(p_2) := \top$ 
15       else
16         foreach  $c \in \theta(p_1)$  do
17           propagate sets  $\{c + 1\} \rightarrow \theta(p_2)$ 
18     foreach  $(p_1 \rightarrow p_2) \in E_P$  do
19       if  $\theta(p_1) == \top$  then
20          $\theta(p_2) := \top$ 
21       else
22         foreach  $c \in \theta(p_1)$  do
23           propagate sets  $\{c - 1\} \rightarrow \theta(p_2)$ 
24     foreach  $(v \xrightarrow{f} p) \in E_{PL}$  do
25       foreach  $o \in \rho(v)$  do
26         propagate sets  $\theta(o, f) \rightarrow \theta(p)$ 
27     foreach  $(p \xrightarrow{f} v) \in E_{PS}$  do
28       foreach  $o \in \rho(v)$  do
29         propagate sets  $\theta(p) \rightarrow \theta(o, f)$ 
30   until no changes;
```

Secondly, the object places and place-typed variables are computed based on the points-to relations. Algorithm 1 shows the procedure. The algorithm begins with propagating all place name nodes to the places of their successors. It repeatedly propagates places along some kinds of edges of the PCG until a fixed point is reached: (1) A place-copy edge of the form $p_1 \rightarrow p_2$ indicates that

	E_{PI}	E_{PC}	E_{loc}	E_H
Complexity	$O(E)$	$O(NE)$	$O(NE)$	$O(N^2E)$
	E_N	E_P	E_{PL}	E_{PS}
Complexity	$O(NE)$	$O(NE)$	$O(N^2E)$	$O(N^2E)$

Table 2. Complexity of Handling Different Constraints

$\theta(p_1) \subseteq \theta(p_2)$, and thus the place set of p_1 is added into the place set of p_2 ; (2) A locality edge from a place-typed variable p to an object o denotes that o is created in p , and thus the place set of p is added into the place locality set of o ; (3) A home edge from o to p means the place locality set of o should be propagated to the places of p ; (4) When a next-place or previous-place edge of the form $p_1 \rightarrow p_2$ is encountered, all the place names in the place set of p_1 shift to the next or previous places and the results are added into the place set p_2 ; (5) When a place-store edge $p \rightarrow v.f$ is encountered, a place-typed variable $o.f$ is created for each object o in the points-to set of v and the place set of p is added into the place set of $o.f$; (6) Similarly, when a place-load edge $v.f \rightarrow p$ is encountered, for each o , the place set of $o.f$ is added into the place set of p . In the worst case, a reference can point to all objects of its type and each place name set is \top .

If the topology structure of places is given, all the place names can be resolved to the places constants. The results can be represented by a locality graph which contains the points-to, object, activity, and place information. In a locality graph, the variable, object and place-typed variable nodes are same as those in the PCG. A locality graph also contains place constant nodes representing place constants in PC . An edge in a locality graph can be

- a points-to edge $v \in V \rightarrow o \in O$,
- an object field edge $o_1 \xrightarrow{f \in F} o_2$, where $o_1, o_2 \in O$,
- a place value edge $p \in P \rightarrow c \in PC$,
- a field value edge $o \in O \xrightarrow{f \in F} p \in P$, or
- an object locality edge $o \in O \xrightarrow{(At)} p \in P$.

For example, Figure 4 shows the locality graphs of the program in Figure 1 for different topology structures.

Convergence. Loops of a PCG generated through the process of PCG construction (e.g., analyzing the recursive method invocations) may lead the place name set of a place-typed variable contain all the place names (i.e., the set is \top), which can slow down the convergence of computing fixed point; even make the computation never terminate. To detect such loops, any loop with respect to a place-typed variable p is unfolded to a path, in which the first and last nodes are p , but p does not appear in the middle. If any unfolded path for p is a L -path and $Prev$ and $Next$ are not matched, we let $\theta(p) = \top$. A L -path is defined

$$\begin{aligned} path &::= match\ path \mid Prev\ path \mid Next\ path \mid \epsilon \\ match &::= Store[f]\ match\ Load[f] \mid match_1\ match_2 \mid \epsilon \end{aligned}$$

where $Prev$ and $Next$ represent the edges in the sets E_P and E_N , respectively; the edge labeled with $Store[f]$ can be a store (resp. place-store) edge; and the edge labeled with $Load[f]$ can be a load (resp. place-load) edge.

For example, the following X10 code brings on a PCG loop $rec(p:Place)\{\dots; async(p)\{\dots\}; rec(p.next())\}$.

The place name set of p is \top due to the L -path $p \xleftarrow{Next} p$.

Complexity. Let N be the number of nodes and E be the number of edges in a PCG. The complexity of the context-insensitive analysis is $O(N^3E)$ in that the complexities of the points-to analysis algorithm and Algorithm 1 are both $O(N^3E)$. The complexity of

```

async(Place(0)) { //a1
  val b1= new B(); //o1
  b1.m(b1.home.next());
}
async(Place(1)) { //a2
  val b2= new B(); //o2
  b2.m(b2.home.next());
}
}

```

```

public class B {
  public var data:T;
  public def m(p:Place):Void{
    async(p.prev()){ //a3
      this.data = ...;
    }
  }
}

```

Figure 6. An Example of X10 Program Analyzed in Different Context-Sensitivities

Algorithm 1 is $O(N^3E)$ because (1) The complexities of solving the constraints of different types are shown in Table 2, where the complexities of propagation operation of copying one points-to set (resp. place set) and adding one place name to a place set are linear and $O(1)$, respectively. (2) The `repeat-until` loop at line 5 needs to execute for $O(N)$ times. (3) The complexity of detecting the loops is $O(N^3)$ because it can be expressed as a CFL-reachability problem [16].

5. Activity-Sensitive Locality Analysis

The precision of Leopard is fair because the interprocedural analysis does not distinguish the activities spawned in different contexts, and thus some place sets are propagated along with some infeasible paths. Figure 6 shows an example of X10 program, where the activities a_1 and a_2 create the objects o_1 and o_2 , respectively, and then invoke the method $B.m()$. In a context-insensitive analysis, this points to the objects o_1 or o_2 and p is either $Place(1)$ or $Place(2)$. Thus the activity a_3 can be spawned at $Place(0)$ or $Place(1)$ after $p.prev()$ is evaluated. Therefore, `this.data=...` is unsafe because `this` may refer to an object whose place is different from that of a_3 . However, in a context-sensitive analysis, we distinguish the case of a_1 spawning a_3 ($[a_1, a_3]$) and a_2 spawning a_3 ($[a_2, a_3]$). In the former case, it can be deduced that `this` only points to o_1 and the place of o_1 is same as that of a_3 . Thus we can infer that `this.data=...` is safe. In the latter case, `this` only points to o_2 and `this.data=...` is also a safe access.

We propose an AS (Activity-Sensitive) locality analysis which adopts an activity sequence to model a context. A context defines a sequence of activities spawned in the last k steps. A context

$$\delta \in \Delta = A^{\leq k}$$

is generated by traversing an ANG, where k is a user-defined constant value.

A context generation algorithm (see Algorithm 2) is adopted in the AS locality analysis for generating all the contexts and binding each context to the activities and methods. At line 7, $(a_0 \xrightarrow{*} a_1 \in A)$ denotes that the activity a_0 directly spawns the child activity a_1 or through a chain of method invocations. At line 9, the new context δ' is generated by $[\delta, a_1]_k$ which represents the k -rightmost truncation of the sequence of $[\delta, a_1]$. At line 19, $(a \xrightarrow{*} m \in M)$ denotes that the method m is reachable with respect to the activity a .

Our AS locality analysis then combines the context information and the activities, methods, objects and variables, to form the new representation

$$\begin{aligned} \delta &\in \Delta, a \in A, m \in M, o \in O, v \in V, p \in P, \\ (\delta, a) &\in A', (\delta, m) \in M', \mu' : \Delta \rightarrow P', \\ (\delta, o) &\in O', (\delta, v) \in V', (\delta, p) \in P', \end{aligned}$$

where the function μ' maps each context δ to a place-typed variable (δ, ep) denoting the places where the activity (resp. methods) of δ is executed. μ' is different from μ in Section 3.2 in that μ' requires all methods of the same context share one place-typed variable,

Algorithm 2: Contexts Generation

```

input : An activity nesting graph G.
output: A context set  $\Delta$ .
1 begin
2    $\Delta$ .clear();
3    $\delta_0 := [\text{root}]$ ;  $\Delta$ .add( $\delta_0$ );  $A'$ .add( $(\delta_0, \text{root})$ );
4    $\text{worklist.add}(\text{root})$ ;
5   while  $\text{worklist.isEmpty}()$  do
6      $a_0 := \text{worklist.remove}()$ ;
7     foreach ( $a_0 \xrightarrow{*} a_1 \in A$ )  $\in G$  do
8       foreach ( $\delta, a_0 \in A'$ ) do
9          $\delta' := [\delta, a_1]_k$ ;
10         $A'$ .add( $(\delta', a_1)$ );
11         $\Delta$ .add( $\delta'$ );
12        if  $\Delta$  is changed then
13           $\text{worklist.add}(a_1)$ ;
14  foreach ( $\delta, a \in A'$ ) do
15    foreach ( $a \xrightarrow{*} m \in M$ )  $\in G$  do
16       $M'$ .add( $(\delta, m)$ );
17  return  $\Delta$ ;

```

but μ maps each method (resp. activity) to a different place-typed variable ep .

The abstract domains are

$$\begin{aligned}
 \rho' &\in (V' \cup (O' \times F_o) \rightarrow \wp(O')), \\
 \sigma' &\in (O' \rightarrow \wp(PN)), \\
 \theta' &\in (P' \cup (O' \times F_p) \rightarrow \wp(PN)).
 \end{aligned}$$

Since each activity or method is bound to some context(s), its statements need to be analyzed under current context. We extend the notation of \models_I and define a relation \models_S

$$(\rho', \sigma', \theta') \models_S s.$$

Figure 7 shows the clauses corresponding to different statement types. Here δ is the current context of the analyzed statements; o^δ , v^δ , and p^δ denote (δ, o) , (δ, v) , and (δ, p) , respectively.

Let a be the activity spawned by the *async* statement in the clause **[async*]**. The statements in a are analyzed under a context $[\delta, a]_k$. $\mu'([\delta, a]_k)$ lookups the place-typed variable under the context $[\delta, a]_k$. Since the child activity can access the variable declared in the parent activity, *CRefEq* requires all the shared variables hold the same place name set (resp. points-to set), which is

$$\forall r \in \text{Acc}(a) : \varphi'((\delta, r), ([\delta, a]_k, r)) \wedge \varphi'(([\delta, a]_k, r), (\delta, r)),$$

where $\text{Acc}(a)$ is a set containing all variables defined in the parent of a and accessed by a , and φ' is defined by

$$\varphi'(r'_a, r'_b) = \begin{cases} \rho'(r'_a) \subseteq \rho'(r'_b) & r'_a, r'_b \in V', \\ \theta'(r'_a) \subseteq \theta'(r'_b) & r'_a, r'_b \in P', \\ \text{false} & \text{otherwise.} \end{cases}$$

φ' handles the parameter passing in **[inv*]**.

[obj*] denotes that our analysis is heap-sensitive in that a static object is created under the current context.

6. Experiments

We have developed a tool to support Leopard. Figure 8 shows the framework of the tool: the X10 compiler (ver. 2.0.3) extracts the AST of an X10 program; the T.J. Watson Libraries for Analysis (WALA) [18] generates its intermediate representation; Leopard

[seq*]	$(\rho', \sigma', \theta')$	$\models_S s_1 s_2$ iff $(\rho', \sigma', \theta') \models_S s_1 \wedge (\rho', \sigma', \theta') \models_S s_2$
[if*]	$(\rho', \sigma', \theta')$	$\models_S \text{if}(\text{cond}) s_1 \text{ else } s_2$ iff $(\rho', \sigma', \theta') \models_S s_1 \wedge (\rho', \sigma', \theta') \models_S s_2$
[async*]	$(\rho', \sigma', \theta')$	$\models_S \text{async}(p) s$ iff $(\rho', \sigma', \theta') \models_S s \wedge \theta'(p^\delta) \subseteq \theta'(\mu'([\delta, a]_k)) \wedge \text{CRefEq}$
[copy*]	$(\rho', \sigma', \theta')$	$\models_S v_1 = v_2$; iff $\rho'(v_2^\delta) \subseteq \rho(v_1^\delta)$
[obj*]	$(\rho', \sigma', \theta')$	$\models_S v = \text{new } C();$ iff $\{o^\delta\} \subseteq \rho'(v^\delta) \wedge \theta'(\mu'(\delta)) \subseteq \sigma'(o^\delta)$
[load*]	$(\rho', \sigma', \theta')$	$\models_S v_1 = v_2.f;$ iff $\forall o' \in \rho'(v_2^\delta), \rho'((o', f)) \subseteq \rho'(v_1^\delta)$
[store*]	$(\rho', \sigma', \theta')$	$\models_S v_1.f = v_2;$ iff $\forall o' \in \rho'(v_1^\delta), \rho'(v_2^\delta) \subseteq \rho'((o', f))$
[ini*]	$(\rho', \sigma', \theta')$	$\models_S p = \text{Place}(i);$ iff $\{i\} \subseteq \theta'(p^\delta)$
[home*]	$(\rho', \sigma', \theta')$	$\models_S p = v.\text{home}();$ iff $\forall o' \in \rho'(v^\delta), \sigma'(o') \subseteq \theta'(p^\delta)$
[here*]	$(\rho', \sigma', \theta')$	$\models_S p = v.\text{here};$ iff $\theta'(\mu'(\delta)) \subseteq \theta'(p^\delta)$
[ass*]	$(\rho', \sigma', \theta')$	$\models_S p_1 = p_2;$ iff $\theta'(p_2^\delta) \subseteq \theta'(p_1^\delta)$
[np*]	$(\rho', \sigma', \theta')$	$\models_S p_1 = p_2.\text{next}();$ iff $\theta'(p_2^\delta) \gg 1 \subseteq \theta'(p_1^\delta)$
[pp*]	$(\rho', \sigma', \theta')$	$\models_S p_1 = p_2.\text{prev}();$ iff $\theta'(p_2^\delta) \ll 1 \subseteq \theta'(p_1^\delta)$
[pt*]	$(\rho', \sigma', \theta')$	$\models_S p = v.f;$ iff $\forall o' \in \rho'(v^\delta), \theta'((o', f)) \subseteq \theta'(p^\delta)$
[ps*]	$(\rho', \sigma', \theta')$	$\models_S v.f = p;$ iff $\forall o' \in \rho'(v^\delta), \theta'(p^\delta) \subseteq \theta'((o', f))$
[inv*]	$(\rho', \sigma', \theta')$	$\models_S r = r_0.m(r_1, \dots, r_k);$ iff $\varphi'(\text{ret}_m^\delta, r^\delta) \wedge \varphi'(r_0^\delta, \text{this}_m^\delta) \wedge \varphi'(r_1^\delta, f_{r_1}^\delta) \wedge \dots \wedge \varphi'(r_k^\delta, f_{r_k}^\delta)$

Figure 7. Activity- and Heap-Sensitive Locality Analysis Specification

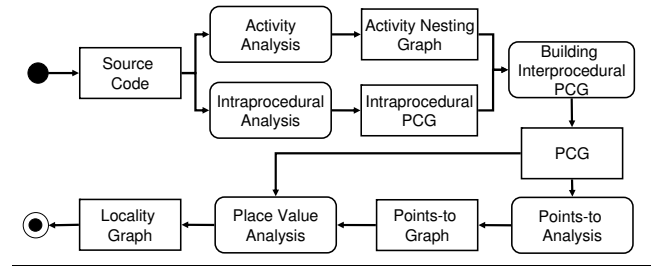


Figure 8. The Framework of Leopard Tool

consumes the intermediate representation and produces the activity nesting graph and the intraprocedural PCGs of all methods and activities; after that, Leopard combines the graphs into an interprocedural PCG and computes the locality graph of the program.

We have conducted two experiments to evaluate the performance, precision and usability of Leopard. We have used 10 benchmark programs from the High-Performance Computing challenge (HPCC) benchmarks, the Java Grande (JG) benchmarks in X10 and the Network-Attached Storage (NAS) benchmarks. Table 3 shows the details of these benchmarks, including the lines of code (LOC), numbers of activities (#Activity), methods (#Method) and method invocations (#Call). Note that X10 language specification varies from version to version, and thus the large scale benchmark programs are not available in our experiments. Leopard also do not completely cover the standard X10 libraries (including 186 classes) in its analysis for cost-effective reason, but just includes all reachable classes in them. In the experiment we ran the tool on a PC (Intel Core™2 Duo CPU 2.40GHz, 2GB memory).

6.1 Performance and Precision

The first experiment was used to evaluate the performance and precision of three analyses: context-insensitive (CI) analysis, one-level AS (activity-sensitive) analysis (i.e., $k = 1$), and two-level AS analysis (i.e., $k = 2$).

Program	LOC	#Activity	#Method	#Call	Suite
sor	123	7	24	21	JG
stream	132	2	10	8	HPCC
sparsemm	345	4	32	25	JG
series	389	3	14	17	JG
crypt	565	2	24	25	JG
moldyn	700	14	36	25	JG
lufact	853	7	24	40	JG
raytrace	1205	13	65	132	JG
mg	1868	57	122	248	NAS
mc	3150	3	83	80	JG

Table 3. Benchmarks

Table 4 shows the sizes (i.e., numbers of nodes/edges) of PCGs and the runtime for each benchmark program. Compared with the CI analysis, the AS analyses have generated some larger scale PCGs and spent more time in solving the PCGs. In the AS analyses, the sizes of PCGs and the corresponding solving time also increased along with the growth of k . One reason is that k determines the variables and objects to be analyzed: the bigger k is, the more clones of variables and objects need to be produced in the analyses.

After investigating the experiment results, we believe that the performance of analyses depends on two factors: the size of the PCG and #Activity. For a program of a small PCG or that of a few activities (e.g., the programs **stream**, **sparsemm**, **series**, and **crypt**), the AS analysis was precise when $k \geq 2$, but the cost was not of significant difference from that of the CI analysis. For a program of many activities or that of a large PCG, the cost increases dramatically along with the increase of k . For example, it took the two-level AS analysis more time in analyzing **mg** than in analyzing **mc**.

In order to evaluate the precision of the analysis, we adopt

- Pct.IO which represents the percentage of the points-to set with only one object in order to measure the precision of the points-to analysis. The closer to 100% the Pct.IO is, the more precise the points-to analysis is.
- Pct.IP which provides the percentage of the place-typed variable with one place name in order to show the possibility of an object or activity residing in one and only one place. The closer to 100% the Pct.IP is, the more precise the locality analysis is.

Figure 9 shows the Pct.IO and Pct.IP of the three analyses. The x-axes of Figures 9(a) and 9(b) represent Pct.IO and Pct.IP, respectively. It can be seen that both Pct.IO and Pct.IP of the AS analyses are higher than those of the CI analysis. The programs **raytrace** and **mg**, analyzed by the CI analysis, are of low Pct.IPs (85.57% and 83.67%). The CI analysis of X10 program is instinctively of low precision in that the points-to sets of the formal parameters of a method called by several callers can grow rapidly, which will be further propagated to other variables and therefore reduce the precision of analysis of place-typed variables. Compared with the CI analysis, the AS analysis improves the precision through distinguishing more activities, objects and method invocations and computing different points-to and locality information for the variables and objects in the activities under different contexts.

We have also found that the precision of the points-to analysis can help improve the evaluation of the place expressions. One reason is that the evaluation of the widely used place expressions in form of $r.home()$ strongly depends on the points-to set of the variable r . A place-typed variable can be propagated through the accesses of place-typed fields in form of $v.p$, the evaluation of which also depends on the points-to set of v .

	SA				CSA		
	PTS	CI	AS		CI	AS	
			k=1	k=2		k=1	k=2
sor	6	6	6	9	0	0	1
stream	5	7	8	11	0	1	2
sparsemm	24	27	35	41	2	2	5
series	7	7	7	7	0	1	1
crypt	17	25	32	40	0	0	3
moldyn	23	27	36	43	0	2	5
lufact	30	37	48	51	1	3	6
raytrace	29	35	42	46	0	4	4
mg	121	145	167	173	7	11	13
mc	43	47	51	51	2	4	7

Table 5. Safe Access Check

6.2 Safety Checking of Instance Field Access

In X10, the instance field $v.f$ can be accessed by activities at place p which is the same place of the object v points to. A static checking of safety can help reduce the overhead of the runtime locality checking. The report of the unsafe accesses can help to find potential errors in X10 programs.

In the second experiment, we study the safety checking of the instance field accesses, which can be classified into two categories:

- Safe Access (SA) which means that the access is safe with whatever the place topology structures are,
- Conditional Safe Access (CSA) which means the access is safe with respect to certain place topology structures.

Table 5 shows the results of checking of the SAs and CSAs for each benchmark program. In the table, we have compared the SAs computed by the CI and AS analyses with the results inferred by the place type system (PTS) [4], and also reported the CSAs computed by the CI and AS analyses.

Table 5 shows that the CI and AS analyses have detected more SAs than PTS has. One reason is that PTS misses some safe accesses due to the limited inference step. That is, the UNIFICATION algorithm of PTS needs to unify the limited depth of field access path for different variables, and any function recursion needs to be unfolded up to the limited depth. Another reason is that PTS omits the place-shift queries (e.g., $p.next()$ and $p.prev()$) and place-typed field accesses (e.g., $v.p$). The AS analyses also have obtained more SAs than the CI analysis has. The possible reason is that the AS analyses can achieve a context-specific name set of one place name for both the activities and the objects accessed. In a CI analysis, the name set is usually rough because it may be the union of these context-specific name sets.

Leopard also has the capability of finding the CSAs in that it models the topology structure of places and place-shift queries and adopts the place name mechanism. Leopard first computes a place name set for each activity and object and then maps the place names to the concrete place based on the specific place topology structures. After that, Leopard decides whether an instance field access is safe with respect to some mappings. For example, in the program **mg**, the statement `mg.results = new BMRResults(...)`; is executed at a place named pn_1 , while `mg` only points to the object at a place named pn_2 . If $pn_1 == pn_2$, the access is safe for any place topology structure. Otherwise, we construct a set

$$S = \{ mp \mid pn_1 \text{ mod } mp == pn_2 \text{ mod } mp \}$$

and check whether the access is safe with respect to the mp -place ($mp \in S$). An empty set denotes that the access is unsafe.

In the study the AS analyses have found more CSAs than the CI analysis has because the AS analysis distinguishes more static objects and activities and computes smaller place name sets for them. In addition, the AS analyses reduce the points-to set size

	Time(ms)			#Node			#Edge		
	CI	AS		CI	AS		CI	AS	
		k=1	k=2		k=1	k=2		k=1	k=2
sor	538	1474	3594	223	701	1325	343	1053	2396
stream	224	258	317	164	183	274	213	260	309
sparsemm	878	2350	3125	576	1463	1768	585	1567	2150
series	2146	6142	8573	1264	2873	4426	1473	3417	4602
crypt	4434	8232	14218	1874	3028	5072	2363	4326	7247
moldyn	5167	29546	55127	2014	10083	19716	2587	13444	23277
lufact	6392	49524	103495	2373	15682	30220	2957	20635	41398
raytrace	9187	80214	221810	2986	21906	63173	3773	27730	79966
mg	13767	445470	5403810	3887	111367	1318121	4971	148490	1734370
mc	15651	31454	61067	4273	11312	20315	5217	14141	25396

Table 4. Performance

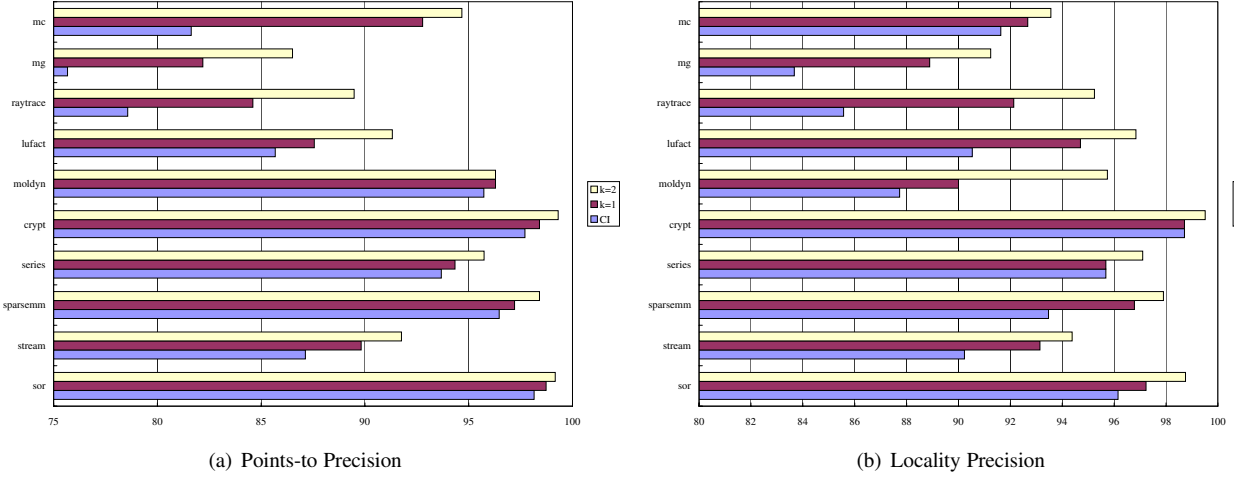


Figure 9. Precision of Solving PCGs

of v in the field access expression $v.f$ and improve the precision of points-to analysis, which will further reduce the complexity of inferring place equivalence conditions.

7. Related Work

In this section, we discuss some work related to points-to analysis and locality analysis of X10 programs.

Points-to Analysis. In the past several years, points-to analysis has been an active research field. A survey of algorithms and metrics for points-to analysis has been given by Hind [9]. Context-sensitivity and flow-sensitivity are two major dimensions of pointer analysis precision. Context-sensitive and flow-sensitive algorithms (CSFS) [6, 12, 20, 21] are usually precise, but are difficult to scale to large programs. Kahlon [11] has proposed a bootstrapping analysis framework that improves the scalability of the context-sensitive and flow-sensitive algorithm. Yu et al. [24] propose a level-by-level algorithm that improves the scalability of context-sensitive and flow-sensitive algorithms. Context-insensitive and flow-insensitive (CIFI) algorithms [3, 17] have the best scalability on the large programs with overly conservative results. Equality-based analysis and inclusion-based analyses have become the two widely accepted analysis styles. By carrying out experiments, Foster et al. [7] compare several variations of flow-insensitive points-to analysis for C, including polymorphic versus monomorphic and equality-based versus inclusion-based. How well the algorithms scale to large programs is another important issue. Trade-off is made between efficiency and precision by various points-to analyses, in-

cluding context-sensitive and flow-insensitive analyses [14, 15, 19] and context-insensitive and flow-sensitive analyses [5, 8, 10].

Different from a pointer analysis, a locality analysis is to analyze not only the points-to relations, but also the places the objects (and activities) reside in. Apart from tracking the points-to information, our PCG also tracks the places of activities and objects. In addition, the traditional points-to analyses occasionally take into account the *place* which is a core concept in the X10 programming model. Different from the traditional context-sensitive analyses, our activity- and heap-sensitive analysis captures the essential constructs of X10 language (e.g., activities and objects). The experiment results show that the activity-sensitive analysis improves the precision of both points-to and locality analyses.

Locality Analysis for X10. Only a few analysis approaches have been proposed for analyzing X10 programs. Agarwal et al. [2] present the algorithm for May-Happen-in-Parallel (MHP) analysis of X10 programs, in which a global place-value numbering is used to analyze X10 place expressions. However, some place expressions still cannot be precisely calculated in a static manner, for example, $v.\text{home}()$ or here cannot be globally assigned. Chandra et al. [4] propose a dependent type system to capture fine-grained locality information. The type system only contains two forms of place expressions: $\text{Place}(i)$ and $v.\text{home}()$, but omits handling the cyclic places in X10 programs. In addition, the type system can calculate the SAs but the CSAs because it can deduce that an object may reside in here , a fixed place or an unknown place. Another limitation of their type system is that the system handles recursion approximately by unfolding the recursion up to some predefined

depth. Leopard overcomes that limitation since it computes a fixed point for an interprocedural PCG. Agarwal et al. [1] present an intraprocedural analysis framework for statically establishing place localities in X10. In the framework, the places and activities are analyzed and the classical thread escape analysis is extended to trace which activities an object can escape. However, how to calculate a place expression containing `p.next()` or `p.prev()` is omitted. Compared with the related work, Leopard deduces a place set for each object in a program aware of the topology structures of places, and thus the analysis can be precise and efficient. We have also conducted some experiments to demonstrate the precision and efficiency of Leopard.

8. Conclusion

In this paper we presented a context-insensitive algorithm and a context-sensitive algorithm for computing the locality information of objects and activities in X10 programs. We also developed a tool to support the Leopard method and conducted some experiments to measure the performance and precision of Leopard. The experimental results show that Leopard produces precise locality information for X10 programs. In addition, we use the results of Leopard to check the safe accesses to instance fields.

In the future, we would like to combine the symbolic execution techniques and Leopard analysis to improve the precision of locality analysis. We would also like to study the code optimizations for X10 programs using our locality analysis results.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable and thorough comments. We would like to thank Feng Xie and Cheng Zhang for their discussions on this work. This work is supported by the National Natural Science Foundation of China (Grant No. 91118004, 61100051, 61272102, 60970009) and Shanghai Key Laboratory of Computer Software Testing & Evaluating (Grant No. SSTL2011.02).

References

- [1] S. Agarwal, R. Barik, V. K. Nandivada, R. K. Shyamasundar, and P. Varma. Static detection of place locality and elimination of runtime checks. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008*, pages 53–74, 2008.
- [2] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007*, pages 183–193, 2007.
- [3] L. Andersen. Program analysis and specialization for the C programming language. DIKU report 94-19, University of Copenhagen, 1994.
- [4] S. Chandra, V. A. Saraswat, V. Sarkar, and R. Bodík. Type inference for locality analysis of distributed data structures. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008*, pages 11–22, 2008.
- [5] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, 1993.
- [6] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
- [7] J. S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the 7th International Symposium on Static Analysis*, pages 175–198, 2000.
- [8] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 226–238, 2009.
- [9] M. Hind. Pointer analysis: haven’t we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.
- [10] M. Hind, M. Burke, P. Carini, and J. deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.
- [11] V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the ACM SIGPLAN 2008 conference on Programming Language Design and Implementation*, pages 249–259, 2008.
- [12] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 56–67, 1993.
- [13] O. Lhoták and L. J. Hendren. Scaling java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 153–169, 2003.
- [14] O. Lhoták and L. J. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proceedings of the 15th International Conference on Compiler Construction*, pages 47–64, 2006.
- [15] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14:1–41, 2005.
- [16] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.
- [17] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.
- [18] WALA. http://wala.sourceforge.net/wiki/index.php/Main_Page.
- [19] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 131–144, 2004.
- [20] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 187–206, 1999.
- [21] R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 1–12, 1995.
- [22] X10. <http://x10.codehaus.org/>.
- [23] K. A. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. N. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. L. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the International Workshop on Parallel Symbolic Computation*, pages 24–32, 2007.
- [24] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th International Symposium on Code Generation and Optimization*, pages 218–229, 2010.